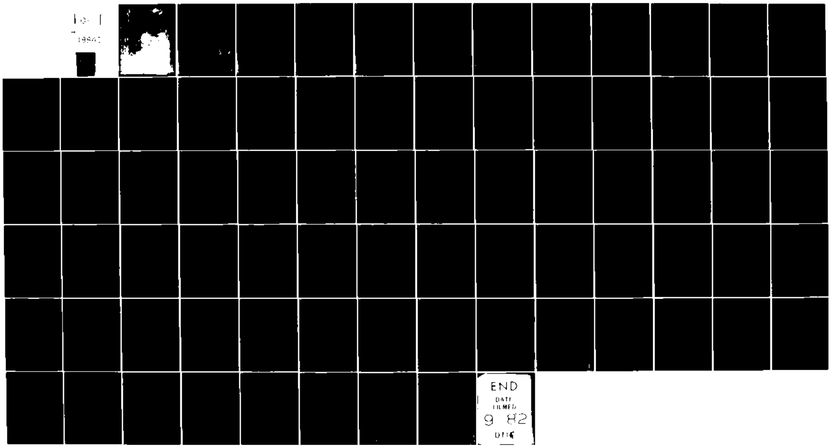
AD-A118 840 GEORGIA INST OF TECH ATLANTA SCHOOL OF INFORMATION A--ETC F/G 9/2
EQUIVALENCE TESTING FOR FORTRAN MUTATION SYSTEM USING DATA FLOW--ETC(U)
DEC 81 A TANAKA N00014-79-C-0231
UNCLASSIFIED GIT-ICS-82/10 NL

1 of 1
1998AT



END
DATE
FILMED
9 82
DTIC

(12)

GIT-ICS-82/10

EQUIVALENCE TESTING FOR FORTRAN MUTATION SYSTEM USING DATA FLOW ANALYSIS*

Akihiko Tanaka

July, 1982

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

*This research was supported in part by ONR Grant No. N00014-79-C-0231 and ARO Grant No. DAAG29-80-C-0120.

EQUIVALENCE TESTING FOR FORTRAN MUTATION SYSTEM USING DATA FLOW ANALYSIS

A THESIS

Presented to

The Faculty of the Division of Graduate Studies

By

Akihiko Tanaka

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Information and Computer Science

Georgia Institute of Technology

December, 1981

ACKNOWLEDGEMENTS

The author would like to thank his thesis advisor, Dr. Richard A. DeMillo for valuable suggestions and thoughtful criticisms, the members of his reading committee, Dr. Richard J. LeBlanc and Dr. N. Jon Livesey, for their helpful comments and suggestions, and his academic advisor, Dr. James Gough Jr., for his assistance with the final manuscript. The author also wishes to express his gratitude to Dr. Frederick G. Sayward of Yale university for his helpful comments. The author is also grateful to the Division of Graduate Studies for the waiver of certain format requirements so that this thesis could be prepared on the PRIME 400 computer system using the Software Tools text editor and text formatter. This research was supported in part by ONR Grant No. N00014-79-C-0231 and ARO Grant No. DAAG29-80-C-0120.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

ABSTRACT

Equivalence Testing for FORTRAN Mutation System

Using Data Flow Analysis

Akihiko Tanaka

67 Pages

Directed by Dr. Richard A. DeMillo

Program mutation is a new approach to program testing, a method designed to test whether a program is either correct or radically incorrect. It requires the creation of a nearly correct program called a mutant from a program P. An adequate set of test data distinguishes all mutants from P by comparing the outputs. Obviously, an equivalent mutant, which performs identically to P, produces the same outputs as those of P. Thus, for adequate data selection, it is desirable that an equivalent mutant be excluded from the testing process. For this purpose, the system equivalence command has been implemented as an equivalent mutant detector. As yet, the command has not been automated. Automatic detection by this command is implemented here as an application of data flow analysis. Algorithms and implementation techniques of data flow analysis are described. Also its application as an automatic detector is described.

CHAPTER I
INTRODUCTION

In conventional program tesing methods, a program is considered as a black box and tested for input cases to execute all statements in the program at least once. Then the outputs produced are checked for correctness. The fundamental question in program testing is:

If a program is correct on a finite number of test cases, is it correct in general ?

Even if the test results are correct, it does not guarantee the absence of errors, or program correctness. Program testing can be used to discover the presence of errors, but not their absence. However, confidence in the reliability of a program can be increased by different testing approaches. For instance, a selection of test cases based on the program structure is more reliable than a random selection [Huan] and symbolic execution is more reliable than execution on numeric data [Howd].

Most testing strategies appeared in 1970's and some are described in [GG] and [Huan]. Program mutation is a relatively new approach to program testing, a method designed to test whether a program is either correct or radically incorrect [DLS]. By radically incorrect is meant that a program contains errors due to grossly misunderstanding the program specification. Many errors may remain undetected even when every statement in a program is executed at least once. On the other hand, if different parts of a program are executed over data which takes into account the kind of errors that can occur in that part of the program, then a significant number of undetected errors will be detected. Mutation testing can construct test data of this type.

Program mutation requires the creation of nearly correct programs called mutant programs (the precise definition of a mutant will be described later). Assuming that a program P performs properly with a set of test data T and a mutant program P' is generated from P, if T distinguishes P and any P', that is, all outputs differ, then T is indeed a comprehensive set of test data; whereas if some P's do not change the test results, but since they might change on augumented test data T', T is inadequate.

To achieve the design goal of program mutation, interactive program mutation systems have been implemented for several different languages such as FORTRAN and COBOL. One version of mutation systems, the Fortran Mutation System (FMS.2), was designed primarily as an experimental device for the mutation research groups at the Georgia Institute of Technology and Yale University. This system produces programs which differ from the original one in very simple ways: a single change is made on a statement. For example, I = I + 1

may be changed as follows:

I = <u>J</u> + 1 ; I was replaced with J

I = I <u>*</u> 1 ; + was replaced with *

I = I + <u>2</u> ; 1 was replaced with 2.

A statement generated by the system as shown above is referred to as a mutant and a program which differs from the original is referred to as a mutant program.

During the course of execution of all mutants created by the system, mutants are classified into three categories: dead, live and equivalent mutant. By a dead mutant is meant that a mutant program is distinguished from the original by either failing to produce any result or producing a different result, whereas by a live mutant is meant that a mutant program produces the same result as the original by testing data sets. An equivalent mutant is defined as one in which the mutant program performs identically to the original, that is, the control-path of this mutant program is equivalent to that of the original.

As mentioned above, a set of test data T is inadequate unless every result of the mutation run is different. In other words, some live mutant programs which should have failed did not. Mutation testing must continue by executing live mutants on augumented test data T'. However, if any equivalent mutant is detected from live mutants before their execution, it is eliminated from a live mutant list. It causes the decrease of mutant runs. Therefore detection of equivalnt mutants plays an important role in improving system performance.

Then the question arises:

How are equivalent mutants detected ?

In the current version of the FORTRAN Mutation System, FMS.2, some equivalent mutants can be detected, though the process is not automatic: a user must provide information to the system equivalence command issued from the terminal. For instance, consider the following example:

```
        I = 1
        J = I + 1.
```

A mutant derived from the above statement is

```
        J = ABS (I) + 1.
```

Since I is greater than zero, I is always equal to ABS(I). Thus, the mutant does not affect the original, that is, this mutant is equivalent. It can be detected with user assistance, i.e., the user executes an equivalence command which tells the system that the value of I is greater than zero. The format of an equivalence command will be discussed in Chapter III. The above example is a very simple case. However, recognition of values of each variable at a certain point in a program, if the program flow is complicated, is difficult and tedious, leading to the omission of variables or mis-setting of their values. Thus, automatic equivalent mutant detectors are desirable.

If detection of equivalent mutants is to be automated, a systematic mechanism is required for its implementation. This mechanism must provide as output the arguments needed for execution of the system equivalence command. As discussed later in Chapter III, the arguments of this command are variables and their values at a given point of a program. Because of these requirements and the ease of applicability and implementation, data flow analysis was chosen.

Prior to the discussion of the system flow of the automatic detector shown in Figure 1.1, data flow analysis and its terminology are briefly described here. Data flow analysis is a static analysis method considering data items, or variables in a program flow graph. Data flow is a control path of a program with information regarding variables. This information contains whether any reference to some variable lies in a control flow or interrupted by another value assigned to the variable. In order to recognize the information, a flow graph of the program is created by partitioning the program into basic blocks. A basic block is defined as a sequence of statements to be executed that have no branches but the first and the last statement of the sequence. Nodes of the flow graph represent basic blocks.

```
           +----------------------+
           | ( source program )   |
           +----------------------+
                      |
                      v
                 +--------+
                 | FMS.2  |
                 +--------+
                      |
                      v
     +--------------------------------+
     | ( intermediate code array )    |
     +--------------------------------+
                      |
                      v
          +----------------------+
          | basic block routine  |
          +----------------------+
                      |
                      v
            +-------------------+
            | interval routine  |
            +-------------------+
                      |
                      v
           +---------------------+
           | data flow routine   |
           +---------------------+
                      |
                      v
               +-------------+
               | ( output )  |
               +-------------+
                      |
                      v
          +----------------------+
          | application routine  |
          +----------------------+
                      |
                      v
               +------------+
               | E command  |
               +------------+
```

Figure 1.1 System Flow of the Automatic Detector

After the creation of the flow graph. the graph is reduced by recognizing an interval that represents a new node of a reduced graph. An 'interval' is a group of nodes whose edges go to nodes inside the group except for edges at the entrance and the exit. Graph reduction, frequently discussed in the rest of this thesis, is defined so that nodes in an interval are merged into a single node and all edges from inside the interval are deleted and new edges to nodes in other intervals are created. A graph is reduced in the above manner until it is irreducible. Finally, the information regarding variables is built by using all flow graphs. Based on this information, data flow analysis can determine at a given node of a program flow graph which variables 'reach' a node and are available at the exit from the node. By 'reach' is meant that a variable is defined at one node and is available at the entrance to another node. The details of data flow analysis will be discussed in the next chapter.

The system as shown in Figure 1.1 was designed and implemented for automatic detection of equivalent mutants. The FORTRAN Mutation System, FMS.2 [BLSD], which includes a scanner, a parser and a code generator, parses FORTRAN source programs and generates intermediate codes. The basic block routine partitions a source program into basic blocks, based on intermediate codes from FMS.2 and creates a flow graph of a source program. The interval routine reduces a flow graph from the basic block routine as much as possible and keeps all reduced flow graphs, from the original graph to the final graph. All graphs generated by the basic block routine and the interval routine are input for data flow analysis. As output, the data flow routine produces variables that reach each node, and variables that are available at the exit from each node. Then an application routine of data flow analysis creates information for an equivalence command from output of data flow analysis. Finally, an equivalence command is issued by the system as if it were issued by the user.

The purpose of this thesis is to detect equivalent mutants by automatically using data flow analysis in a system that creates data for an equivalence command and that issues this command automatically as if issued by the user. A detailed discussion of data flow analysis is given in Chapter II which includes algorithms and implementation techniques. Chapter III covers applications of data flow analysis, how to use output from data flow analysis. The conclusion in Chapter IV contains the results of the system described in Chapter II and III. Appendix A contains an example of data flow analysis and Appendix B contains a sample run on FMS.2 with a description of equivalent mutants.

CHAPTER II

DATA FLOW ANALYSIS

This chapter describes algorithms and implementation techniques of data flow analysis, including the basic block routine and the interval routine. The idea is derived from [BC] and [AC]. Data flow analysis, a technique employed for the system, examines data-flows other than control-flows in a program. This analysis is based on information collected at compile time. The FORTRAN parser on the FMS.2 system generates intermediate codes from a source program which are independent of a target machine and more related to a statement in a source program than machine codes. Those intermediate codes are saved in a file during program parsing. Because of ease of access to the file and so that the data flow analysis system can be independent of the parser, a flow graph is created by using these codes.

The data flow analysis system consists of three phases: basic block, interval and data flow. Data flow analysis requires a flow graph of a source program and graphs reduced from the original graph. Two pre-processors of the data flow routine, the basic block routine and the interval routine, generates these graphs. Figure 1.1 in Chapter I depicts the system flow of data flow analysis.

The basic block routine partitions a source program into basic blocks and creates a flow graph. The graph generated by this routine is passed to the interval routine.

This routine examines flows of the original graph and merges any possible node in the graph into a single node. Then all unnecessary edges are deleted to reduce a graph. Graph reduction continues as far as it can. All redued graphs are saved and passed to the data flow routine. This routine examines data flows of a source program using all information passed from two routines discussed above. The analysis of data flow is based on flow graphs to see if values assigned to variables are changed.

Prior to the further discussion of data flow analysis, graph representation in memory is described here (the details will be discussed later in SECTION 2.2). A node has two different kinds of pointers that represent edges to and edges from the node: one pointer points to a destination node for an edge and the other points to a source node. If a node has branches, that is, if there are multiple edges leaving the node, all destination nodes are linked together. Also all source nodes are linked if more than one edge come to the node.

Section 2.1 describes the algorithms of the basic block routine, the interval routine and the data flow routine, and Section 2.2 describes their implementation techniques.

SECTION 2.1 Algorithms

This section discusses the details of the algorithms of the three phases of the data flow system described above. The description of the basic block routine describes how to distinguish corresponding intermediate codes for each FORTRAN statement and how to use these codes for partition. The next sub-section discusses the algorithm of the interval routine, i.e., detection of intervals and reduction of flow graphs. The main phase of the data flow analysis system is discussed in the last sub-section. The data flow routine describes the process of determination of available variables at each basic block.

BASIC BLOCK ROUTINE

The basic block routine referenced in Figure 1.1 partitions a FORTRAN source program into basic blocks by using intermediate code generated by FMS.2, and then creates a flow graph of the source program. Since the flow graph directly corresponds to a source program, it is called the original graph or the lowest order graph. This graph is passed to the interval routine for reduction.

A basic block is a sequence of statements to be executed which has only one entrance or none (the first basic block does not have an entrance) and which has the only exit or none (the basic block which ends with a RETURN statement or a STOP statement does not have an exit). Thus a basic block is considered to be a node of a flow graph.

The remainder of this sub-section discusses seven basic rules of partition on a FORTRAN statement. After recognition of a statement, a basic rule is applied on the statement. According to these rules, incoming and outgoing edges to and from a node are determined. Also, the beginning and the end of a node are determined. The basic rules applied on a statement are:

1) A statement with a label is the entrance to a basic block.

Prior to this statement, the last basic block must be closed, if it is not; a new basic block beginning with this statement will be opened because it is possible to jump into this statement by a GOTO statement. A basic block cannot be entered in the middle.

2) An IF statement is an exit from a basic block.

i) logical IF statement.

IF (expression EXPR) statement S

The intermediate codes for a logical IF statement are as follows:

```
IOP 0
|
|   codes for expression EXPR
|
TRF index of the next statement
|
|   codes for statement S
|
```

The first code for an IF statement is IOP and only the IF statement starts with IOP. A logical IF statement is distinguished from an arithmetic IF statement by including a key code TRF; while codes for an arithmetic IF statement include a key code AIF. A jump address is set when the statement S is a GOTO statement. The S itself is considered to be a basic block if S is a RETURN statement, a STOP statement or an ASSIGNMENT statement. It is not necessary to consider the ASSIGNMENT statement a basic block except when it is a part of an IF statement. The IF statement is partitioned into two basic blocks: the IF (expression EXPR) itself is a basic block and the then-part statement S is also a basic block because a value assigned to a variable in the ASSIGNMENT statement changes if the boolean expression EXPR is evaluated to be true (see Example II below). The following two examples illustrate the above two cases. In Example I, the IF statement is not partitioned into two basic blocks because the execution of the IF statement does not affect any values of variables (I, J and K). On the other hand, the execution of the IF statement in Example II might cause changes of a value of K. In this case, the value of K depends on path selection in the flow. So the IF statement is partitioned into two basic blocks (2) and (3).

```
    Example I                     basic block

    I = 1                         -- { 1 }
    J = 2                         -- { 1 }
    IF ( I .EQ. J ) GOTO 10       -- { 1 }
    K = 3                         -- { 2 }
 10 STOP                          -- { 3 }
    END
```

The flow graph of the above example is:

```
{ 1 }---+
  |     |
  |     |
  v     |
{ 2 }   |
  |     |
  |     |
  v     |
{ 3 }<--+
```

```
Example II                basic block

I = 1                     -- { 1 }
J = 2                     -- { 1 }
IF ( I .EQ. J ) K = 3     -- { 2 } & { 3 }
STOP                      -- { 4 }
END
```

The flow graph of the above example is:

```
{ 1 }
  |
  |
  v
{ 2 }---+
  |     |
  |     |
  |     v
  |   { 3 }
  |     |
  v     |
{ 4 }<--+
```

ii) arithmetic IF statement.

```
IF ( expression EXPR ) L1,L2,L3
```

The intermediate codes for an arithmetic IF statement are as follows:

```
IOP 0
|
|   codes for expression EXPR
|
AIF 0
LABEL index of L1 in STMT array
LABEL index of L2 in STMT array
LABEL index of L3 in STMT array
```

As mentioned above, a key code AIF distinguishes an arithmetic IF statement from a logical IF statement. Three jump addresses are set according to the values of the three LABEL codes.

3) A DO statement is the entrance to a basic block.

```
DO 10 I = expression EXPR1, EXPR2, EXPR3
```

The intermediate codes for a DO statement are as follows:

```
IDNT index of I in the symbol table
SEPR 0
|
|   codes for expression EXPR1
|
SEPR 0
|
|   codes for expression EXPR2
|
SEPR 0
|
|   codes for expression EXPR3
|
DOST index of statement 10
```

A statement whose code starts with IDNT might be either a DO statement or an ASSIGNMENT statement. To find out which it is, check the last code for this statement: the DO statement always ends with a key code DOST, while the ASSIGNMENT statement always ends with a key code ASSIGN. The DO loop is partitioned into basic blocks just like the other statements. Nested DO loops are handled with a stack which pushes down DOST with value, which is a DO loop label (in the above case, 10 is a label), whenever DOST appears and pops it up whenever a corresponding CONTINUE statement appears (i.e., 10 CONTINUE appears in the above case).

4) A GOTO statement is an exit from a basic block.

i) simple GOTO statement.

```
GOTO L1
```

The intermediate code for a GOTO statement is as follows:

```
BR index of target statement
```

The only code generated for a simple GOTO statement is BR. A jump address is set.

ii) computed GOTO statement.

```
GOTO (L1,L2,L3,...Ln) expression EXPR
```

The intermediate codes for a computed GOTO are as follows:

```
|
|     codes for expression EXPR
|
CGOTO 0
LABEL index of L1
LABEL index of L2
LABEL index of L3
|             .
|             .
|             .
LABEL index of Ln
```

A statement whose last code is LABEL is either a GOTO statement or an arithmetic IF statement. To determine this, check the code which precedes the first LABEL code: if it is CGOTO, then the statement is a computed GOTO statement; otherwise, it is an arithmetic IF statement (a code preceding the first LABEL must be AIF). There are as many jump addresses set as LABEL codes. Only a computed GOTO statement generates more than three jump addresses, while an arithmetic IF statement generates three, a logical IF statement and a CONTINUE statement two.

5) A CONTINUE statement is an exit from a basic block.

```
10 CONTINUE
```

The intermediate code for a CONTINUE statement is as follows:

```
       CONT 0
```

The only code generated for a CONTINUE statement is CONT. A CONTINUE statement indicates the end of a DO loop, corresponing to a DO statement (in the above case it correspondes to DO 10 ..... statement). Whenever a CONTINUE statement appears and a stack for a nested DO loop is not empty, a statement label of a CONTINUE statement is compared with a DO statement label contained in the stack. If labels are identical, the stack pops up an entry.

6) A STOP statement is itself a basic block.

The intermediate code for a STOP statement is as follows:

```
       STP 0
```

The only code generated for a STOP statement is STP. The STOP statement is considered to be an independent basic block because it is convenient to make an exit node of a flow graph and it is necessary to make an exit from a basic block when the STOP statement is the then-part of an IF statement.

7) A RETURN statement itself is a basic block.

The intermediate code for a RETURN statement is as follows:

```
RET 0
```

The only code generated for a RETURN statement is RET. The RETURN statement is treated exactly as a STOP statement.

INTERVAL ROUTINE

This sub-section discusses the interval analysis algorithm. Using a flow graph passed from the basic block routine, this routine performs detection of an interval that is a group of nodes whose immediate predecessors are from the node of the same group except for a header node. It also performs reduction of flow graphs. The graph generated by a source program is referred to as the lowest order graph. The last graph created by the interval routine, which is no longer reducible, is referred to as the highest order graph. A graph reduced from another graph is said to be higher in order than the graph to be reduced. For example, if G2 is reduced from G1, G2 is higher than G1; whereas G1 is lower than G2. All flow graphs reduced by this routine are passed to the data flow routine.

By a header node is meant an entrance to an interval. By a predecessor is meant a node which comes into another node in a flow graph, whereas by a successor is meant a node which goes out from another node (see Figure 2.1). An interval is a new node for the next reduced graph.

```
{ 1 }    --- predecessor of node { 2 }
  |
  |
  v
{ 2 }
  |
  |
  v
{ 3 }    --- successor of node { 2 }
```

Figure 2.1 Predecessor and Successor of a node

The first process of determining an interval is to select the root node (the entry point) of a flow graph as a header node. Then each successor node of the header node is examined whether it comprises an interval. A node to be added to the interval should be one whose predecessors all come from inside the interval. A node that fails to comprise the interval is considered to be a candidate of a header node of another interval. When all successors of each node in the interval are examined but no more nodes can be added to the interval, the next header node is selected to recognize another interval. The process continues until all nodes are in intervals. The detailed algorithm of detection of intervals in a PASCAL-like language is shown in Algorithm 2.1.

```
procedure interval-routine ;
var
   H : array of header ;
   I : array of interval ;
begin
   add the first node of the flow graph to H ;
   for all h in H do
      begin
         add h to I[h] ;
         for all i in I[h] do
            for all j in { successors of i } do
               if ( all predecessors of j in I[h] ) then
                  add j to I[h] ;
         for all i in I[h] do
            for all j in { successors of i } do
               if not ( j in I[h] ) then add j to H ;
      end
end ; -- end of interval-routine
```

Algorithm 2.1 Interval Routine

The algorithm for reducing graphs is rather simple. An interval will represent a node of the next higher order graph, i.e., the graph reduced from a graph which is processed for detecting intervals. After detection, edges between nodes within an interval are deleted and edges from outside the interval are changed to point to the header node of the interval; edges to other nodes outside the interval are changed to point to the header nodes of each interval.

DATA FLOW ROUTINE

This sub-section discusses the data flow routine, the main and last routine of the data flow analysis system. All graphs generated by the basic block routine and the interval routine are collected for this routine to examine data flows. The routine determines the variable that reaches each node and the variable that is available at the end of each node by tracing each variable on a flow graph. All information generated by the routine is passed to the system described in the next chapter, Chapter III.

The data flow analysis algorithm described below consists of two phases.. A definition, frequently discussed below, is a statement that assigns a value to a variable, replacing a previous value; that is, a definition is an assignment statement. All sets of definitions referred to below are A, D, DB, DOUT, P, PB and R. These contain definitions for each node in all flow graphs (see also the declaration part of Algorithm 2.2). Phase I determines P, the definitions preserved on some path through the interval to the exit, and D, the definitions in the interval that may be available, depending on the path. PB, a set of definitions preserved in some node, and PD, a set of locally available definitions, are computed by using values of P and D. Any definition that reaches a node is said to be preserved by the node. A locally available definition is the last definition of a variable within a node. For example,

```
I := 1 ; --- (1)
J := 2 ; --- (2)
K := 3 ; --- (3)
I := 4 ; --- (4)
```

Assuming that the above four assignment statements comprise a node, locally available definitions are (2), (3) and (4) since variable I is re-assigned in statement (4) (statement (1) is 'killed' by statement (4)). R[i] denotes a set of definitions that can reach a node i from inside the interval. Phase I is performed in the following order: from the original graph to the highest order graph. On the other hand, phase II is performed in the reverse order of phase I. A[i], appearing in phase II, denotes a set of available definitions on the edge i. Phase II determines A, the definitions available at the exit from a node, and R, the definitions that reach the node from other nodes, by using the results of phase I, i.e., PB and PD.

The details of the algorithm are shown in Algorithm 2.2. Note that + and * denote UNION and INTERSECTION over sets respectively.

```
procedure data-flow-analysis ;
var
   A : set of available definitions ;
   D : set of definitons in the interval
       that may be availabe on the exit ;
   DB : set of locally available definitions ;
   DOUT : set of definitions reaches from outside ;
   P : set of definitions preserved on some path
       through the interval to the exit ;
   PB : set of definitions preserved ;
   R : set of definitions that reach nodes ;
   G : set of flow graph ;
begin
   initialization ;
   phaseI ;
   if ( DOUT <> { } ) R[1] := DOUT ;
   phaseII ;
end ; -- end of data-flow-analysis

procedure phaseI ;
var
   g : graph-number ;
   h : header-node-number ;
   x : exit-edge-number ;
   l : edge-from-inside ;
   p : input-edge-number ;
begin
n := number of graphs generated by interval-routine ;
for g:= 1 to n - 1 do
   begin
      if ( g > 1 ) then
         for all i in { edges in G[g] } do
            begin
               x := corresponding exit edge
                    in G[g-1] to i ;
               h := header node of I[h] with x ;
               PB[i] := P[x] ;
               DB[i] := (R[h] * P[x]) + D[x] ;
            end ;
      for all h in { header nodes in G[g] } do
         begin
            for all i in { exit edges of h } do
               begin
                  P[i] := PB[i] ;
                  D[i] := DB[i] ;
               end ;
            R[h] := empty-set ;
            for all l in { edges which enter h
                                from inside I[h] } do
               R[h] := R[h] + D[l] ;
         end ;
      for all j in { non-header nodes in G[g] } do
         begin
            PP := empty-set ;
            DP := empty-set ;
            for all p in { input edges of node j } do
               begin
                  PP := PP + P[p] ;
                  DP := DP + D[p] ;
               end ;
            for all i in { exit edges of node j } do
               begin
                  P[i] := PP * PB[i] ;
                  D[i] := (DP * PB[i]) + DB[i] ;
               end ;
         end ;
   end ;
end ; -- end of phaseI

procedure phaseII ;
var
   g : graph-number ;
   h : header-node-number ;
   p : input-edge-number ;
begin
n := number of graphs generated by interval-routine ;
for g:= n - 1 downto 1 do
   begin
      for all i in { nodes in G[g+1] } do
         begin
            h := header node in G[g]
                 which i represents in G[g+1] ;
            R[h] := R[h] + R[i] ;
         end ;
      for all h in { header nodes in G[g] } do
         begin
            for all i in { exit edges of h } do
               A(i) := (R[h] * PB[i]) + DB[i]
            for all j in { non-header nodes in I[h] } do
               begin
                  R[j] := empty-set ;
                  for all p in { input edges to j } do
                     R[j] := R[j] + A[p] ;
                  for all i in { exit edges to j } do
                     A[i] := (R[j] * PB[i]) + DB[i] ;
               end ;
         end ;
   end ;
end ; -- end of phaseII
```

Algorithm 2.2 Data Flow Analysis

SECTION 2.2 Implementation

This section discusses implementation techniques of the data flow analysis system shown in Figure 1.1. The system was divided into two phases, the data flow analysis system and the applications system. The data flow analysis system consists of three routines: the basic block routine, the interval routine and the data flow routines, as shown in Figure 1.1. The algorithms discussed in the last section have been implemented. The applications system will be discussed in the next chapter.

The initial discussion in this section concerns interrelationships among routines, that is, which routines a routine calls and from which routines a routine can be called. The former is defined as a child routine and the latter is defined as a parent routine. In the following discussion, the relation between a parent and a child routine is referred to as a hierarchy. A figure of the hierarchy depicts a parent-child relationship. An arrow --> in the figure goes from a parent to a child routine. The second discussion deals with the data structure of the data flow analysis system.

Figure 2.2 depicts the hierarchy of the data flow analysis system. The basic block routine, the interval routine and the data flow routine are BSCBLK, INTRVL and DTAFLW respectively. The DRIVER routine called by FMS.2 controls these three routines and WRITVL of the data flow analysis system. First, DRIVER calls BSCBLK which partitions a source program into basic blocks and produces a flow graph; it then calls INTRVL repeatedly until a graph cannot be reduced. DRIVER can recognize when a graph can no longer be reduced, by comparing the number of nodes in the previous graph with that of the graph last generated. Finally, DTAFLW is called to produce a set AVLSET of available definitons for each node and a set RCHSET of definitions that reach each node. For tracing and debugging purposes, WRITVL prints a header node and other non-header nodes for each interval. BSCBLK, INTRVL and DTAFLW calls routines. Each routine is described further below.

```
                    +---------+
                    | DRIVER  |
                    +---------+
                         |
    +--------------------------------------+
    |           |            |             |
    |           |            |             |
    v           v            v             v
+---------+ +---------+ +---------+ +---------+
| BSCBLK  | | INTRVL  | | WRITVL  | | DTAFLW  |
+---------+ +---------+ +---------+ +---------+
```

Figure 2.2 Hierarchy of the Data Flow Analysis System

In order to create a flow graph, BSCBLK requires the mechanism for generating edges. For this purpose, BSCBLK calls the SETOUT routine when an arithmetic IF statement or a computed GOTO statement appears. Intermediate codes for both statements include LABEL code. SETOUT sets the statement number to be executed next; it will be changed to a basic block number after the creation of the original flow graph. Such a basic block number represents a destination node of an edge. WRINTR prints internal forms (LIMS array, Symbol Table, Statement array and CODE array [BH]) generated by FMS.2.

It would be helpful to discuss FMS.2 for understanding interfaces between FMS.2 and a newly implemented system. The FORTRAN Mutation System (FMS.2) is an interactive system for testing the completeness of a set of test data on a FORTRAN source program. The user is requested to give the name of the program being tested, mutant operators which are to be applied, and test cases which are used. During program parsing, information is saved in the arrays including those discussed above and mutants are created according to operators specified by the user. Then the system executes each mutant on test data, eliminating dead mutants. The results of mutant runs are reported back to the user. He can request various reports and summries. He can continue the experience if necessary.

```
          +--------+
          | BSCBLK |
          +--------+
              |
       +-------------+
       |             |
       |             |
       v             v
  +--------+     +--------+
  | SETOUT |     | WRINTR |
  +--------+     +--------+
```

Figure 2.3 Hierarchy of the BSCBLK Routine

A flow graph generated by the BSCBLK routine should be reduced for examining data flows. To do this, DRIVER, the control routine of the data flow analysis system, calls INTRVL after BSCBLK. INTRVL partitions a flow graph into intervals which are nodes in the next higher order flow graph. This algorithm was described in Algorithm 2.1 in the last section. Whenever this routine is called by DRIVER, it creates the next higher order graph to the extent that a graph can be reduced. An edge of a flow graph is deleted simply by deleting a node number which represents the destination of the edge. FNDNOD is called to examine whether a node belongs to some interval. This routine is useful to check whether all immediate predecessors are already in the interval. After finding one interval, INTRVL selects the next unprocessed header node, calling SHLHDR which selects the smallest number among unprocessed header nodes.

```
           +--------+
           | INTRVL |
           +--------+
               |
        +-------------+
        |             |
        |             |
        v             v
   +--------+    +--------+
   | FNDNOD |    | SMLHDR |
   +--------+    +--------+
```

Figure 2.4 Hierarchy of the INTRVL Routine

After the execution of the BSCBLK routine and the INTRVL routine, all information requested by the DTAFLW routine is generated. Then DRIVER calls DTAFLW (for the details see Algorithm 2.2 in the last section). DTAFLW consists of INTDF, PHASE1, INTRCH and PHASE2 as shown in Figure 2.5. INTDF initializes DB and PB sets discussed in the previous section. Performance of PHASE1 and PHASE2 is described in Algorithm 2.2 above. DTAFLW calls INTDF, PHASE1, INTRCH and PHASE2 in order; each of which is called once. INTRCH, called between PHASE1 and PHASE2, initializes the R set for the root of the highest order graph, if a definition reaches the program from outside such as a parameter of a subroutine in FORTRAN. Variables set by DATA statements are treated as parameters. Since all four routines called by DTAFLW call some other routines, details of each routine will be described below.

```
                    +---------+
                    | DTAFLW |
                    +---------+
                         |
         +------------------------------------+
         |           |           |            |
         |           |           |            |
         v           v           v            v
    +---------+  +---------+  +---------+  +---------+
    | INTDF   |  | PHASE1 |  | INTRCH |  | PHASE2 |
    +---------+  +---------+  +---------+  +---------+
```

Figure 2.5 Hierarchy of the DTAFLW Routine

INTDF stands for INiTialization for the Data Flow analysis routine. The first subroutine called from INTDF, FNDPRM finds external definitions and are set by DATA statements. The former are parameters of subroutines passed from outside the program. If parameters are found, BBNUM and ENTPNT fields in DEFTBL are set; other fields, however, are set simply to zero (i.e., variables are undefined). On the other hand, if the latter is found, BBNUM, ENTPNT and TYPKND in DEFTBL and its value are set. Variables set by DATA statement are constant since they are read-only variables in FMS.2. In order to distinguish the former from the latter, BBNUM field is set to -1 for the former and 0 for the latter.

```
                          +---------+
                          | INTDF   |
                          +---------+
                               |
       +-----------------------------------------+
       |              |              |              |
       |              |              |              |
       v              v              v              v
  +---------+    +---------+    +---------+    +---------+
  | FNDPRM  |    | DEFPB   |    | DEFDB   |    | WRITDT  |
  +---------+    +---------+    +---------+    +---------+
       |                             |
       |                  +-------------+
       v                  |             |
  +---------+             |             |
  | FMS.2   |             v             v
  |routines |        +---------+   +---------+
  | ADDRES  |        | FNDASN  |   | SETDB   |
  |   DP    |        +---------+   +---------+
  |  REL    |                           |
  | SCTYPE  |                     +-------------+
  +---------+                     |             |
                                  |             |
                                  v             v
                             +---------+   +---------+
                             | FNDCON  |   | FNDID   |
                             +---------+   +---------+
                                  |
                                  |
                                  v
                             +---------+
                             | FMS.2   |
                             |routines |
                             |   DP    |
                             |  REL    |
                             +---------+
```

Figure 2.6 Hierarchy of the INTDF Routine

DB and PB are defined from the original flow graph by calling DEFDB and DEFPB respectively. Further, DEFDB calls FNDASN to find an assignment statement and, if found, calls SETDB to determine a DB set. To find an assignment statement that assigns a constant value to a variable, FNDCON is called. If a constant value is found on the right-hand side, that is, the expression on the right-hand side is a simple constant, the value of the constant and its type (integer, real or logical) are set in the table of definitions (DEFTBL). FNDID performs recognition of locally available definitions. Definitions appearing more than once within a node are detected and killed by the FNDID routine except for the last appearance. Also, DEFTBL is updated if a definition is killed. Using the results from DEFDB, PB set is determined; for each node PB is all definitions in DEFTBL minus DB for each node. WRITDT prints contents of DEFTBL, identifier, value and its value if any.

PHASE1 determines PB, DB and R sets for each node in the original graph to the highest order graph. PB and DB are computed by using values of P and D in the previous graph. If a node is a header, P and D are equal to PB and DB respectively; however, in the case where a node is not a header, the unions over P and D for all input edges to the node are used to compute P and D for each exit edge. DEFPRE defines this union over P, called DFPSET, and DEFDEF defines the union over D, called DFDSET. For each header node, R is defined by the union over D for edges that reach the header node from inside the interval. DEFRCH defines the above union. During processing in PHASE1 three set operations are utilized: union, intersection and assignment which are referred to as UNION, INTRSC and TRNSFR respectively in

PHASE1.

```
                      +--------+
                      | PHASE1 |
                      +--------+
                           |
       +-------------------------------------+
       |           |            |             |
       |           |            |             |
       v           v            v             v
  +--------+  +--------+  +--------+  +-----------+
  | DEFPRE |  | DEFDEF |  | DEFRCH |  |    set    |
  +--------+  +--------+  +--------+  | operators |
       |           |            |     |   TRNSFR  |
       |           |            |     |   UNION   |
       v           v            v     |   INTRSC  |
  +--------+  +--------+  +--------+  +-----------+
  | UNION  |  | UNION  |  | UNION  |
  +--------+  +--------+  | TRNSFR |
                          +--------+
```

Figure 2.7 Hierarchy of the PHASE1 Routine

PHASE2 is the final step in the DTAFLW routine, determining the A set for each exit and the R set for each node in the graph from the highest order to the original graph. The final output of data flow analysis , A and R in the original flow graph, is produced by PHASE2. For a non-header node, R is defined as the union over the A set for all input edges to the node which is computed by calling the DEFAVL routine. The three set operations are utilized in PHASE2 as well as PHASE1.

```
                +---------+
                | PHASE2  |
                +---------+
                     |
            +---------------+
            |               |
            |               |
            v               v
       +--------+     +------------+
       | DEFAVL |     |    set     |
       +--------+     | operators  |
            |         |   TRNSFR   |
            |         |   UNION    |
            v         |   INTRSC   |
       +--------+     +------------+
       | UNION  |
       | TRNSFR |
       +--------+
```

Figure 2.8 Hierarchy of the PHASE2 Routine

DATA STRUCTURE

This sub-section discusses the data structure of the data flow analysis system. All arrays and variables in the following discussion, declared as COMMON arrays and variables in FORTRAN, are used by the routines described above. Data structure will be described in PASCAL-like language for understandability and readability.

1) BLKTBL

```
BLKTBL is array[1..BTSIZE] of
   record
      LAST:1..MAXST: -- last stmt of a basic block
      PTROG:1..OGSIZE: -- pointer to OUTGO
      PTRIC:1..ICSIZE: -- pointer to INCOM
      PTRRCH:1..SETSIZ; -- pointer to RCHSET
      HDRNO:1..BTSIZE; -- header node of interval
      PRVHDR:1..BTSIZE; -- corresponing header node
                        -- in previous graph
   end;
```

BKLTBL contains basic blocks consisting of the above seven fields. LAST field contains the last statement of a basic block. PTROG points to OUTGO array which contains a successor node and its information. Similarly, PTRIC points to INCOM array for predecessor nodes. PTRRCH points to RCHSET, a set of definitions that can reach a node (R set in the above algorithm). HDRNO contains a header node of the interval to which a node belongs. On the other hand, PRVHDR contains a corresponding header node in the previous graph (lower order graph), unless the node is in the original graph. This field is useful to determine the PB and DB sets in PHASE1 of the data flow analysis routine.

2) GRAPH

GRAPH is array[1..GRSIZE] of 1..BTSIZE;

A subscript of GRAPH represents a graph order and an element of GRAPH points to an element of BLKTBL which represents the last node of a flow graph. IGRAPH points to the highest order graph.

3) OUTGO

```
OUTGO is array[1..OGSIZE] of
   record
      BBNUM:1..BTSIZE; -- basic block(= node) number
      NEXT:1..OGSIZE; -- pointer to next element in OUTGO
      EDGENO:1..OGSIZE; -- edge number
      PTRDB:0..SETSIZ; -- pointer to DBSET
      PTRPB:0..SETSIZ; -- pointer to PBSET
      PTRDEF:0..SETSIZ; -- pointer to DEFSET
      PTRPRE:0..SETSIZ; -- pointer to PRESET
      PTRAVL:0..SETSIZ; -- pointer to AVLSET
   end;
```

OUTGO, pointed from PTROG field in BLKTBL, represents exit edges to successor nodes with linked-list structure by using NEXT field as a pointer to the next element, if more than one successor node, the content of BBNUM, exist. An exit edge number in EDGENO field is unique in all graphs. PTRDB, PTRPB, PTRDEF, PTRPRE and PTRAVL point to DBSET, PBSET, DEFSET, PRESET and AVLSET respectively.

4) INCOM

```
INCOM is array[1..ICSIZ] of
   record
      BBNUM:1..BTSIZE; -- basic block(= node) number
      NEXT:1..ICSIZE; -- pointer to next element in INCOM
   end;
```

INCOM, pointed from PTRIC field in BLKTBL, represents input edges from a node, the content of BBNUM field in INCOM. The structure of INCOM is also linked-list with NEXT field as a pointer to the next element.

5) NEWNOD

```
NEWNOD is array[1..BTSIZE] of 0..BTSIZE;
```

A subscript of NEWNOD is identical to a node number; whereas an element of NEWNOD represents a corresponding node in the next higher order graph to the node. Reduction of a flow graph requires deletion of exit edges. That is recognized by comparing the element of NEWNOD for a node with that for a successor node. If both are identical, the exit edge is deleted from a flow graph.

6) HEADER

```
HEADER is array[1..HDRSIZ] of
   record
      NODENO:1..BTSIZE; -- header node number
      PTRINT:1..INTSIZ; -- pointer to INTER
   end;
```

During interval analysis, HEADER is used to keep header nodes. The first field of HEADER, NODENO, contains a header node which is either processed or unprocessed by the INTRVL routine; pointer IHDR indicates the currently processing header node and pointer HDRTOP indicates the last header node recognized by the INTRVL routine. Processed header nodes have some nodes which belong to the same interval as a header node. Of these nodes, which are kept in INTER discussed below, the last recognized one is pointed by the second field of HEADER, PTRINT.

7) INTER

INTER is array[1..INTSIZ] of 1..BTSIZE;

INTER contains interval nodes, linked with HEADER. Elements of INTER, in sequence between the element pointed by the previous HEADER.PTRINT and by the current HEADER.PTRINT, belong to the same interval.

8) Sets for data flow analysis

```
DBSET is array[1..SETSIZ] of 0..DTSIZE; -- DB set
PBSET is array[1..SETSIZ] of 0..DTSIZE; -- PB set
DEFSET is array[1..SETSIZ] of 0..DTSIZE; -- D set
PRESET is array[1..SETSIZ] of 0..DTSIZE; -- P set
RCHSET is array[1..SETSIZ] of 0..DTSIZE; -- R set
AVLSET is array[1..SETSIZ] of 0..DTSIZE; -- A set
DFPSET is array[1..SETSIZ] of 0..DTSIZE; -- union over P(p)
                                        -- p: all input edges
DFDSET is array[1..SETSIZ] of 0..DTSIZE; -- union over D(p)
                                        -- p: all input edges
```

The structure of all sets is identical; the element pointed by a pointer in BLKTBL or in OUTGO contains the number of elements of a set for each node or exit edge. The elements following this element are elements of a set, representing definition identifiers discussed below. The following example illustrates the structure of PBSET.

```
OUTGO
|   |
|---|
|   |
|---|     PTRPB
| --|----------------+
|---|                |
|   |      PBSET     v
|---|      ------------------------------------
|   |       | | | |3|1|4|7| | | | | | | | | |
|---|      ------------------------------------
|   |
```

The element pointed by PTRPB, 3 represents the number of elements of PBSET for some exit edge. Three elements of this edge are the numbers following 3: 1,4,7. PTRDB, PTRPB, PTRDEF, PTRPRE and PTRAVL in OUTGO point to DBSET, PBSET, DEFSET, PRESET and AVLSET respectively and PTRRCH in BLKTBL points to RCHSET. These sets are saved for further use. On the other hand, DFDSET and DFPSET, which represent the union over D and P respectively for all input edges, are computed each time D and P are determined for each exit edge of a non-header node. The first element of these two sets contains the number of elements of sets.

9) DEFTBL

```
DEFTBL is array[1..DTSIZE] of -- definition table
   record
      BBNUM:1..BTSIZE; -- basic block numner
      ENTPNT:1..MAXSYM; -- entry point
      ARYON:-1..MAXSUB; -- array indicator
      TYPKND:0..4; -- type kind
      FSTNUM:1..MAXST; -- first appearance of id
      LSTNUM:1..MAXST; -- last appearance of id
   end;
```

DEFTBL contains information about definitions, consisting of six fields: BBNUM, ENTPNT, ARYON, TYPKND, FSTNUM and LSTNUM. A definition appears in the basic block number BBNUM. ENTPNT represents an entry point to Symbol Table for a variable, the left hand side of an assignment statement. If the variable is a simple variable, 0 is set in ARYON; if an array element and its subscript is kown at compile time, a subscript is set; otherwise, -1 is set. TYPKND indicates the type of a constant, if the right hand side of an assignment is a simple constant, that is, without any arithmetic operators. type INTEGER, REAL, LOGICAL and DOUBLE PRECISION are 1, 2, 3 and 4 respectively; in the other cases, the value of TYPKND is 0. FSTNUM field contains a statement number where a variable appears for the first time within the basic block BBNUM; while LSTNUM contains a statement number where a variable appears last within basic block BBNUM. These fields are useful to determine lower and upper limits for an equivalence command (see Chapter III). Note that a definition identifier is identical to a subscript of DEFTBL array.

10) INTVAL, REALVL LGCVAL and DBLVAL

```
INTVAL is array[1..DTSIZE] of INTEGER; -- INTEGER constant
REALVL is array[1..DTSIZE] of REAL; -- REAL constant
LGCVAL is array[1..DTSIZE] of LOGICAL; -- LOGICAL constant
DBLVAL is array[1..DTSIZE] of DOUBLE PRECISION;
                                  -- DOUBLE PRECISION constant
```

These arrays contain a value of the right-hand side of an assignment if it is a simple constant. A subscript of these arrays corresponds to a definition identifier.

CHAPTER III

APPLICATIONS OF DATA FLOW ANALYSIS

This chapter describes applications of data flow analysis and its implementation for the FORTRAN Mutation System.

Based on the information developed in data flow analysis there are some useful applications to detect an equivalent mutant. Such applications are dead code detection, recognition of loop invariants, constant propagation and invariant propagation. If any node, for instance,is not connected to the rest of the flow graph, all mutants generated from this node do not affect the output from the original program. These mutants can be considered to be equivalent mutants as well as logically equivalent ones. For more detailed discussions of applications to testing mutant equivalence, see [ABDLS] and [DLS].

Some techniques of equivalence detection described in [DLS] have already been implemented on the FMS.2 system as an equivalence testing post-processor called Equivalence command. Since a large number of mutants can be eliminated on the first testing run, this processor is run after the mutants have been executed on the data.

On the current FMS.2 system, there are a variety of commands available to displaying status information after the mutant execution. For example, the CE command displays a program listing with equivalent mutants. The equivalent command (E), however, is rather special because it detects equivalent mutants instead of displaying the results of testing. The command format is as follows:

```
E STA stmt_no - stmt_no
> variable_list absolute_value
or
> variable_list EQ constant
```

where stmt_no - stmt_no specifies which statements are under consideration and variable_list is a list of variables separated by commas. The possible values of absolute_value are:

```
POS,NONZ   POS   NEG,NONZ   NEG   NONZ
```

These parameters mean greater than zero, greater than or equal to zero, less than zero, less than or equal to zero and not equal to zero respectively. The detailed explanation of the E command format will be found in [BHS]. The remainder of this chapter will discuss the algorithm of setting arguments of the E command, that is, two stmt_no's, variable_list and absolute_value shown in the above format, its implementation and an example of equivalent mutants detected in a given program.

The applications system is described in the following section. For a node without any definitions, all incoming definitions are available for an equivalence detecting command. However, a node with definitions is treated differently; all reaching definitions which define the same variable as is defined in the node are available from the entrance to the node to the first appearance of the definition. Assuming that the same variable is defined twice in the node, the first definiiton is killed by the second one. Thus, between the first and the second definition, the first one is available. Before the first definition, all reaching definitions sharing a common variable are available and between the second definition and the exit of the node, the second definition is available. All other definitions, which do not define the same variable as defined in the node, are available from the entrance to the exit of the node. The details of algorithm is shown in Algorithm 3.1.

```
procedure EITPRT ;
var
   i : 1..BTSIZE; -- node
   j : 1..OGSIZE; -- exit edge from node i
   s1 : 1..MAXST; -- lower limit of stmt for E command
   s2 : 1..MAXST; -- upper limit of stmt for E command
   DB is array[1..SETSIZ] of 0..MAXST;
                    -- DB set for each edge
   R is array[1..SETSIZ] of 0..MAXST;
                    -- R set for each node
   SAME is array[1..SETSIZ] of 0..MAXST;
                    -- set for same id
begin
if ( DB[j] = { } ) then
   begin -- no assignment statement in node i
      s1 := first stmt of node i ;
      s2 := last stmt of node i ;
      ECMND(R[i],s1,s2) ;
   end
else -- assignment stmt in node i
   for all Xi in DB[j] do -- X is any variable
      begin
         SAME := {Xk | Xk in R[i]} ;
         if ( SAME <> { } ) then
            begin -- X reaches node i from other nodes
               s1 := first stmt of node i ;
               s2 := (stmt with first X in node i) - 1 ;
               ECMND(SAME,s1,s2) ;
            end ;
         s1 := (stmt of Xi) + 1 ;
         s2 := last stmt of node i ;
         ECMND(Xi,s1,s2) ;
         if ( (R[i]-SAME) <> { } ) then
            begin
               s1 := first stmt of node i ;
               s2 := last stmt of node i ;
               ECMND(R[i]-SAME,s1,s2) ;
            end ;
      end ;
end ;
```

Algorithm 3.1 The Applications Systems

Note: procedure ECMND executes an equivalence detecting command. The first parameter is a set of definitions and the second and third one are the lower and upper limit of a statement respectively.

Consider the following example:

```
          |                |
          v                |
     +--------+            v
  {a}| I:=    |       +--------+
     | J:=    |    {b}| I:=    |
     +--------+       +--------+
          |#a              |#b
          |                |
          +----------------+
                  |
                  |
                  v
          +----------------+
       {c}|   |    ---(w)  |
          |   |            |
          | I:=    ---(x)  |
          |   |            |
          |   |            |
          | I:=    ---(y)  |
          |   |            |
          |   |    ---(z)  |
          +----------------+
                  |#c
                  |
                  v
```

Definitions reaching to node {c} are Ia and Ja from node {a} and Ib from node {b} (Note: Xi denotes a definition of a variable X defined in node i). Thus,

R[c] = {Ia,Ja,Ib}

Definitions defined in node {c} are two Ic's but the first definition is killed by the second one. Thus,

DB[c] = {Ic}

In the above algorithm, SAME denotes a set of definitions which reach a node and are defined in the node, that is, a definition which shares the common variable and reaches the node and/or defined in the node. Since I is defined in node {c},

SAME = {Ia,Ib} for node {c} and Ic

Definitions in SAME, Ia and Ib, are available between (w) and (x), the first apprearance of I, instead of (y). On the other hand, Ja is available from (w) through (z) because J is not defined in node {c}. Ic is available between (y)+1 and (z).

The hierarchy of the applications system is shown in Figure 3.1. The INTRFC routine, called by the DISPLY routine in FMS.2, interfaces between the FMS.2 system and the data flow analysis applications system. Each application is called from INTRFC (only EITPRT is the currently available application and further applications will be extended and called from INTRFC). The algorithm discussed above has been implemented as the EITPRT routine. The TRNSFR routine is called, when RCHSET is assigned to another set. SETRLN sets arguments for the equivalence detecting command as if they were set from the terminal. RESULT set passed from EITPRT to SETRLN contains definitions applied for equivalence detecting command. One of the arguments of this command is a value of a variable, if it is known; otherwise, POS, NEG, NONZ or combinations of these (POS and NONZ, NEG and NONZ) are set. POS, NEG and NONZ stand for POSitive, NEGative and NON-Zero values respectively (see [BHS] for an equivalence command). As an internal value for the argument in this applications system, the following values are assigned:

1: value is greater than zero

2: value is less than zero

3: value is greater than or equal to zero

4: value is less than or equal to zero

5: value is non-zero

6: value is zero

7: value is LOGICAL .TRUE.

8: value is LOGICAL .FALSE.

9: value is known (INTEGER or REAL)

0: value is unknown.

If more than one definition with a common variable reach a node, the above value is computed by using Parameter Kind Decision Table(PKTBL) shown in Table 3.1. The first column and first row in PKTBL are parameter kinds for two different definitions which define a common variable.

	1	2	3	4	5	6	7	8
1	1	5	3	0	5	3	0	0
2	5	2	0	4	5	4	0	0
3	3	0	3	0	0	6	0	0
4	0	4	0	4	0	6	0	0
5	5	5	0	0	5	0	0	0
6	3	4	6	6	0	6	0	0
7	0	0	0	0	0	0	7	0
8	0	0	0	0	0	0	0	8

Table 3.1 Parameter Kind Decision Table

The FINDPK routine is called by EITPRT to determine a parameter kind for a definition. This INTEGER FUNCTION routine receives a definition and returns a parameter kind of the definition. The DFECMD routine actually executes an equivalence detecting command using information set by SETRLN. This routine is similar to the routine in FMS.2 which executes the same command, but arguments for DFECMD are set automatically instead of by manual operation. The WRTINF routine traces the execution of an equivalence command, displaying its arguments on the terminal. The SAMEID routine is called by EITPRT to examine whether there are any definitions sharing a common variable between a set of definitions which reach a node and a set of definitions defined in the node.

```
                        +---------+
                        | INTRFC  |
                        +---------+
                             |
                             |
                             v
                        +---------+
                        | EITPRT  |
                        +---------+
                             |
          +----------------------------------------+
          |             |             |             |
          |             |             |             |
          v             v             v             v
     +---------+   +---------+   +---------+   +---------+
     | TRNSFR  |   | SETRLN  |   | SAMEID  |   | SETRLN  |
     +---------+   +---------+   +---------+   +---------+
                        |                           |
                +--------------+                    |
                |              |                    v
                v              v               +---------+
           +---------+    +---------+          | FINDPK  |
           | FMS.2   |    | WRTINF  |          +---------+
           |routines |    +---------+
           | SUBCLA  |         |
           | EMUCMP  |         |
           | PTMIB   |         v
           | PRCLST  |    +---------+
           +---------+    | FMS.2   |
                          |routines |
                          | PRNAME  |
                          | PRNUM   |
                          | PRCHAR  |
                          | TYPEBF  |
                          +---------+
```

Figure 3.1 Hierarchy of the Applications System

INFEQU, which contains the information for an equivalence detecting command, has the following data structure:

```
INFEQU is array[1..INFSIZ] of
   record
      DTINDX:1..DTSIZE; -- definition identifier
      PARKND:0..9; -- parameter kind
   end;
```

DTINDX field contains a definition identifier of DEFTBL and PARKND contains a parameter kind. These two fields are determined by the SETRLN routine and used with the lower and upper limit of a statement by the DFECMD routine. After executing an equivalence detecting command, this information is discarded.

To make clear the above discussion, consider the following FORTRAN program:

```
      SUBROUTINE ASSIGN
      INTEGER I,J,K
      INPUT OUTPUT I,J
      OUTPUT K
      DATA K/3/
   10 I = 2                  -- (1)
      J = 1                  -- (2)
   20 IF (J .LT. 0) I = 3    -- (3) & (4)
   40 J = I + K              -- (5)
      I = 1                  -- (6)
      I = 2                  -- (7)
      J = I + 2              -- (8)
      RETURN                 -- (9)
      END                    -- (10)
```

The mutation system creates 157 mutants for the above program. After the first testing run, 51 mutants were killed and 106 mutants remained as a live mutant. Then the EITPRT routine produced the following E commands and issued them:

```
>E STA  2 - 2
 >I EQ 2

>E STA  1 - 2
 >K EQ 3

>E STA  3 - 3
 >K EQ 3
 >I EQ 2
 >J EQ 1

>E STA  4 - 4
 >K EQ 3
 >J EQ 1

>E STA  5 - 5
 >I POS,NONZ

>E STA  8 - 8
 >I EQ 2

>E STA  8 - 8
 >K EQ 3.
```

EITPRT creates commands for each basic block as shown above. The arguments of these commands are equivalent to the following:

1) K=3 between statement (1) and statement (8)

2) I=2 between statement (2) and statement (3)

3) J=1 between statement (3) and statement (4)

4) I>0 in statement (5)

5) I=2 in statement (8).

Since the data flow analysis routine can not evaluate a boolean expression, the valuable I holds integer value 2 or 3 after the IF statement. Thus, the case 4) appears as shown above. Tnose five cases can be easily analyzed from the above FORTRAN program.

After the execution of E commands created by EITPRT, 16 equivalent mutants were detected. Some of them are:

IF (J .LE. 0) I = 3 for statement (3)

J = I + 3 for statement (5)

J = ABS I + 2 for statement (8).

The complete sample run of this program can be found in Appendix B.

CHAPTER IV

CONCLUSIONS

Data flow analysis recognizes available variables in each basic block and their values if known at compile time. Thus, the more information collected at compile time, the more precise the analysis is. Variables, whose initial values are not assigned at compile time or at the entry point to the program, are undefined until their values are assigned. In experiences with a newly implemented system, the system equivalence command detects an equivalence mutant automatically, based on information produced by data flow analysis.

In the current FORTRAN mutation system, twenty two mutant operators are used. Of these, Scalar for Constant Replacement, Constant for Scalar Replacement, Relational Operator Replacement and Absolute Value Insertion can be affected by the new system. To create a mutant, if a variable is replaced with the value recognized by data flow analysis, this mutant is equivalent. Similarly, if a constant is replaced with a variable whose value is recognized by data flow analysis and equal to the constant, this mutant is also equivalent. If a value of a variable is recognized as greater than or equal to zero, the absolute value of the variable is equal to the original value. Also, this mutant is detected as an equivalent mutant. Suppose that a variable holds less than zero as its value, a relational operator LT is then logically equivalent to LE from a program-flow point of view.

Those mutants created by the above four mutant operators can be detected automatically by the data flow analysis with an equivalence command.

APPENDIX

A. EXAMPLE OF DATA FLOW ANALYSIS

```
        G[1]

         |Y:=
         v
        {1}X:=
         |
         +<---------+
         |#1        |
         v          |
    +---{2}---+     |#5
    |         |     |
    |#2       |#3   |
    v         v     |
   {3}Y:=    {4}----+
    |         |X:=
    |#4       |#6
    v         v
  exit1     exit2
```

```
          G[2]

          |Y:=
          v
         {1}
          |
          |#7
          v
     +---{5}---+
     |         |
     |#8       |#9
     v         v
   exit1     exit2

   {5} = {2,3,4}
```

```
          G[3]

          |Y:=
          v
     +---{6}---+
     |         |
     |#10      |#11
     v         v
   exit1     exit2

   {6} = {1,2,3,4}
```

PHASE I

initialization:

edge #	PB	DB
1	{Y0,Y3}	{X1}
2	{X1,X4,Y0,Y3}	empty
3	{X1,X4,Y0,Y3}	empty
4	{X1,X4}	{Y3}
5	{Y0,Y3}	{X4}
6	{Y0,Y3}	{X4}

G[1]:
edge #1
P[1] = PB[1] = {Y0,Y3}
D[1] = DB[1] = {X1}

edge #2
P[2] = PB[2] = {X1,X4,Y0,Y3}
D[2] = DB[2] = { }

```
edge #Y3
P[3] = PB[3] = {X1,X4,Y0,Y3}
D[3] = DB[3] = { }


edge #4
P[4] = PP * PB[4]
     = P[2] * PB[4]
     = {X1,X4,Y0,Y3} * {X1,X4}
     = {X1,X4}
D[4] = (DP * PB[4]) + DB[4]
     = (D[2] * PB[4]) + DB[4]
     = ({ } * {X1,X4}) + {Y3}
     = {Y3}


edge #5
P[5] = PP * PB[5]
     = P[3] * PB[5]
     = {X1,X4,Y0,Y3} * {Y0,Y3}
     = {Y0,Y3}
D[5] = (DP * PB[5]) + DB[5]
     = (D[3] * PB[5]) + DB[5]
     = ({ } * {Y0,Y3}) + {X4}
     = {X4}


edge #6
P[6] = PP * PB[6]
     = P[3] * PB[6]
     = {X1,X4,Y0,Y3} * {Y0,Y3}
     = {Y0,Y3}
D[6] = (DP * PB[6]) + DB[6]
     = (D[3] * PB[6]) + DB[6]
     = ({ } * {Y0,Y3}) + {X4}
     = {X4}



G[2]:
1) Define PB and DB
edge #7
PB[7] = P[1] = {Y0,Y3}
DB[7] = (R[1] * P[1]) + D[1]
      = ({ } * {Y0,Y3}) + {X1}
      = {X1}
```

```
edge #8
PB[8] = P[4] = {X1,X4}
DB[8] = (R[2] * P[4]) + D[4]
      = ({X4} * {X1,X4}) + {Y3}
      = {X4,Y3}


edge #9
PB[9] = P[6] = {Y0,Y3}
DB[9] = (R[2] * P[6]) + D[6]
      = ({X4} * {Y0,Y3}) + {X4}
      = {X4}


ii) Define P and D
edge #7
P[7] = PB[7] = {Y0,Y3}
D[7] = DB[7] = {X1}


edge #8
P[8] = PP * PB[8]
     = P[7] * PB[8]
     = {Y0,Y3} * {X1,X4}
     = { }
D[8] = (DP * PB[8]) + DB[8]
     = (D[7] * PB[8]) + DB[8]
     = ({X1} * {X1,X4}) + {X4,Y3}
     = {X1,X4,Y3}


edge #9
P[9] = PP * PB[9]
     = P[7] * PB[9]
     = {Y0,Y3} * {Y0,Y3}
     = {Y0,Y3}
D[9] = (DP * PB[9]) + DB[9]
     = (D[7] * PB[9]) + DB[9]
     = ({X1} * {Y0,Y3}) + {X4}
     = {X4}



G[3]:
i) Define PB and DB
edge #10
PB[10] = P[8] = { }
DB[10] = (R[6] * P[8]) + D[8]
       = ({ } * { }) + {X1,X4,Y3}
       = {X1,X4,Y3}
```

```
edge #11
PB[11] = P[9] = {Y0,Y3}
DB[11] = (R[6] * P[9]) + D[9]
       = ({ } * {Y0,Y3}) + {X4}
       = {X4}
```

PHASE II

```
initialization: R[6] = {Y0}



G[3]:
i) Define R
node {6}
R[6] = R[6] + R[1] = {Y0} + { } = {Y0}


ii) Define A
edge #10
A[10] = (R[6] * PB[10]) + DB[10]
      = ({Y0} * { }) + {X1,X4,Y3}
      = {X1,X4,Y3}


edge #11
A[11] = (R[6] * PB[11]) + DB[11]
      = ({Y0} * {Y0,Y3}) + {X4}
      = {Y0,X4}



G[2]:
i) Define R
node {1}
R[1] = R[1] + R[6] = { } + {Y0} = {Y0}


ii) Define A
edge #7
A[7] = (R[1] * PB[7]) + DB[7]
     = ({Y0} * {Y0,Y3}) + {X1}
     = {X1,Y0}
```

```
edge #8
R[5] = A[7] = {X1,Y0}
A[8] = (R[5] * PB[8]) + DB[8]
     = ({X1,Y0} * {X1,X4}) + {X4,Y3}
     = {X1,X4,Y3}


edge #9
A[9] = (R[5] * PB[9]) + DB[9]
     = ({X1,Y0} * {Y0,Y3}) + {X4}
     = {X4,Y0}



G[1]:
i) Define R
node {1}
R[1] = R[1] + R[1] = { } + {Y0} = {Y0}


node {2}
R[2] = R[2] + R[5] = {X4} + {X1,Y0} = {X1,X4,Y0}



ii) Define A
edge #1
A[1] = (R[1] * PB[1]) + DB[1]
     = ({Y0} * {Y0,Y3}) + {X1}
     = {X1,Y0}


edge #2
A[2] = (R[2] * PB[2]) + DB[2]
     = ({X1,X4,Y0} * {X1,X4,Y0,Y3}) + { }
     = {X1,X4,Y0}


edge #3
A[3] = (R[2] * PB[3]) + DB[3]
     = ({X1,X4,Y0} * {X1,X4,Y0,Y3}) + { }
     = {X1,X4,Y0}


edge #4
R[3] = A[2] = {X1,X4,Y0}
A[4] = (R[3] * PB[4]) + DB[4]
     = ({X1,X4,Y0} * {X1,X4}) + {Y3}
     = {X1,X4,Y3}
```

```
edge #5
R[4] = A[3] = {X1,X4,Y0}
A[5] = (R[4] * PB[5]) + DB[5]
     = ({X1,X4,Y0} * {Y0,Y3}) + {X4}
     = {X4,Y0}

edge #6
A[6] = (R[4] * PB[6]) + DB[6]
     = ({X1,X4,Y0} * {Y0,Y3}) + {X4}
     = {X4,Y0}
```

B. EXAMPLE OF A SAMPLE RUN

```
OK, run>seg fms.3
              EXPER - EXPERIMENTAL MUTATION TESTING SYSTEM

                TYPE "?<RET>" AT ANY TIME FOR ASSISTANCE

WHAT IS THE NAME OF THIS EXPERIMENT?
*assign
INITIAL EXPERIMENT
ENTER FILE NAMES
)assign
FILE: ASSIGN
ASSIGN
WHAT TYPES OF MUTANTS DO YOU WANT TO CREATE FOR ASSIGN
*all
       157 MUTANTS CREATED FOR ASSIGN
              NODE #     NEW #
                1          6
                2          6
                3          6
                4          6
                5          6
              HEADER NODE     PTR TO INTER
                   1              5
                   INTERVALS
                       1    2    3    4    5
              NODE #     NEW #
               6           7
              HEADER NODE     PTR TO INTER
                   6              1
                   INTERVALS
                       6
          DEFINITION TABLE
ID #  NODE # SYMTAB ENTRY ARYON  TYPKND  FSTNUM  LSTNUM
   1      0     102         0       1       0       0
VALUE(INTEGER):    3
   2     -1      93         0       0       0       0
   3     -1      84         0       0       0       0
   4      1      84         0       1       1       1
VALUE(INTEGER):    2
   5      1      93         0       1       2       2
VALUE(INTEGER):    1
   6      3      84         0       1       4       4
VALUE(INTEGER):    3
   7      4      93         0       0       5       8
   8      4      84         0       1       6       7
VALUE(INTEGER):    21885
---------- GRAPH #  1 ----------
NODE # =  1
          LAST STMT #   2
```

```
                    POINTER TO OUTGO  1
                    NODE #  2      EDGE #  1
                    DBSET
                        4    5
                    PBSET
                        1
                    DEFSET
                        4    5
                    PRESET
                        1
                    AVLSET
                        1    4    5
                    POINTER TO INCOM  0
                    RCHSET
                        1    2    3
                    PREVIOUS HEADER #  0      HEADER NODE #  1
          NODE # =  2
                    LAST STMT #   3
                    POINTER TO OUTGO  2
                    NODE #  3      EDGE #  2
                    DBSET
                         EMPTY
                    PBSET
                        1    2    3    4    5    6    7    8
                    DEFSET
                        4    5
                    PRESET
                        1
                    AVLSET
                        1    4    5
                    NODE #  4      EDGE #  3
                    DBSET
                         EMPTY
                    PBSET
                        1    2    3    4    5    6    7    8
                    DEFSET
                        4    5
                    PRESET
                        1
                    AVLSET
                        1    4    5
                    POINTER TO INCOM  1
                      1
                    RCHSET
                        1    4    5
                    PREVIOUS HEADER #  0      HEADER NODE #  1
          NODE # =  3
                    LAST STMT #   4
                    POINTER TO OUTGO  4
                    NODE #  4      EDGE #  4
                    DBSET
```

```
                 6
             PBSET
                 1    2    5    7
             DEFSET
                 5    6
             PRESET
                 1
             AVLSET
                 1    5    6
             POINTER TO INCOM  2
               2
             RCHSET
                 1    4    5
             PREVIOUS HEADER #  0     HEADER NODE #  1
     NODE # =  4
             LAST STMT #   8
             POINTER TO OUTGO  5
             NODE #  5     EDGE #  5
             DBSET
                 7    8
             PBSET
                 1
             DEFSET
                 7    8
             PRESET
                 1
             AVLSET
                 1    7    8
             POINTER TO INCOM  3
               2
               3
             RCHSET
                 1    4    5    6
             PREVIOUS HEADER #  0     HEADER NODE #  1
     NODE # =  5
             LAST STMT #   9
             POINTER TO OUTGO  0
             POINTER TO INCOM  5
               4
             RCHSET
                 1    7    8
             PREVIOUS HEADER #  0     HEADER NODE #  1
     DO YOU WANT TO ADD A NEW TEST CASE ?
     #y
     TYPE VALUES FOR VARIABLES  I J
      1 2
     TEST CASE NUMBER     1
     PARAMETERS ON INPUT
     I = 1
     J = 2
     PARAMETERS ON OUTPUT
```

```
K = 3
J = 4
I = 2
THE PROGRAM TOOK         47 STEPS TO EXECUTE
PLEASE VERIFY THAT THE TEST CASE IS CORRECT
*y
DO YOU WANT TO ADD A NEW TEST CASE ?
*n
APPLYING TEST CASE NUMBER      1
157 REMAINING LIVE MUTANTS
    25% OF MUTANTS EXECUTED.  143/  157 LIVE
    50% OF MUTANTS EXECUTED.  135/  157 LIVE
    75% OF MUTANTS EXECUTED.  123/  157 LIVE
   100% OF MUTANTS EXECUTED.  106/  157 LIVE
               >E STA   2 -  2
                >I EQ 2
               >E STA   1 -  2
                >K EQ 3
               >E STA   3 -  3
                >K EQ 3
                >I EQ 2
                >J EQ 1
               >E STA   4 -  4
                >K EQ 3
                >J EQ 1
               >E STA   5 -  5
                >I POS,NONZ
               >E STA   8 -  8
                >I EQ 2
               >E STA   5 -  8
                >K EQ 3

WHAT STATUS INFORMATION DO YOU WANT TO SEE?
*ce prog all


LISTING THE PROGRAM UNIT "ASSIGN"
   WITH SPECIFIED EQUIV MUTANTS
       SUBROUTINE ASSIGN
       INTEGER    K, J, I
       OUTPUT K
       INPUT OUTPUT J, I
       DATA K/3/
10     I = 2
       J = 1
20     IF(J .LT. 0) I = 3

$48$    IF(J .LT. 0) I = K
$59$    IF(1 .LT. 0) I = 3
$112$   IF(J .LE. 0) I = 3
```

```
$119$  IF(ABS J .LT. 0) I = 3
$121$  IF(ZPUSH J .LT. 0) I = 3

40     J = I + K

$68$    J = I + 3
$122$  J = ABS I + K
$124$  J = ZPUSH I + K
$125$  J = I + ABS K
$127$  J = I + ZPUSH K

       I = 1
       I = 2
       J = I + 2

$55$    J = I + I
$69$    J = 2 + 2
$131$  J = ABS I + 2
$133$  J = ZPUSH I + 2
$134$  J = ABS (I + 2)
$136$  J = ZPUSH (I + 2)

       RETURN
        END

WHAT STATUS INFORMATION DO YOU WANT TO SEE?
*m prog all
MUTANT ELIMINATION PROFILE FOR ALL PROGRAMS
MUTANT TYPE                TOTAL  DEAD          LIVE          EQUIV
CONSTANT REPLACEMENT         14    4   28.6%  10 71.4%    0   0.0%
SCALAR VARIABLE REPLACEM     22   10   45.5%  12 54.5%    0   0.0%
SCALAR FOR CONSTANT REP.     21    5   23.8%  14 66.7%    2   9.5%
CONSTANT FOR SCALAR REP.     15    3   20.0%   9 60.0%    3  20.0%
SOURCE CONSTANT REPLACEM      9    2   22.2%   7 77.8%    0   0.0%
UNARY OPERATOR INSERTION     16    5   31.3%  11 68.8%    0   0.0%
ARITHMETIC OPERATOR REPL     14    5   35.7%   9 64.3%    0   0.0%
RELATIONAL OPERATOR REPL      7    0    0.0%   6 85.7%    1  14.3%
ABSOLUTE VALUE INSERTION     18    2   11.1%   6 33.3%   10  55.6%
STATEMENT ANALYSIS            4    3   75.0%   1 25.0%    0   0.0%
STATEMENT DELETION            7    3   42.9%   4 57.1%    0   0.0%
RETURN STATEMENT REPLACE      8    7   87.5%   1 12.5%    0   0.0%
DATA STATEMENT ALTERATIO      2    2  100.0%   0  0.0%    0   0.0%

WHAT STATUS INFORMATION DO YOU WANT TO SEE?
*no
DO YOU WANT TO CONTINUE THE EXPERIMENT?
*n
```

REFERENCES

[ABDLS] A. T. Acree, T. J. Budd, R. A. DeMillo, R. J. Lipton and F. G. Sayward, "Mutation Analysis," 1979.

[AC] F. E. Allen and J. Cocke, "A Program Data Flow Analysis," CACM 19,(3) 1976 pp. 137-147.

[BC] W. A. Barrett and J. D. Couch, Compiler Construction:Theory and Practice, Science Research Associates, 1979.

[BH] T. A. Budd and R. L. Hess, "Exper Implementation Notes."

[BHS] T. A. Budd, R. L. Hess and F. G. Sayward, "User's Guide for EXPER: Mutation Analysis System," 1980.

[BLSD] T. A. Budd, R. J. Lipton, F. G. Sayward and R. A. DeMillo, "The Design of a Prototype Mutation System for Programming Testing," The Proceedings of National Computer Conference, June 5-8, 1978, Anaheim, California, U.S.A.

[DLS] R. A. DeMillo, R. J. Lipton and F. G. Sayward, "Papers on Program Testing," 1979.

[GG] J. B. Goodenough and S. L. Gerhart, "Toward a Theory of Test Data Selection," IEEE Transactions on Software Enginnering, Vol. SE-1, No. 2, June 1975.

[Howd] W. E. Howden, "An Evaluation of the Effectiveness of Symbolic Testing," IEEE Tutorial: Software Testing & Validation Techniques, 1978.

[Huan] J. C. Huang, "An approach to Program Testing," Computing Surveys, Vol. 7, No. 3, September 1975.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER GIT-ICS-82/10 ✓	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Equivalence Testing for Fortran Mutation System Using Data Flow Analysis		5. TYPE OF REPORT & PERIOD COVERED Interim Technical
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Akihiko Tanaka		8. CONTRACT OR GRANT NUMBER(s) ONR-N00014-79-C-0231 ARO-DAAG29-80-C-0120
9. PERFORMING ORGANIZATION NAME AND ADDRESS School of Information and Computer Science Georgia Institute of Technology Atlanta, Georgia 30332		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS [illegible] Res[illegible]		12. REPORT DATE December, 1981
		13. NUMBER OF PAGES 67 + v
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) unclassified
		15a. DECLASSIFICATION DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) [illegible]		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES THE VIEW, OPINIONS, AND/OR FINDINGS CONTAINED IN THIS [illegible]		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) program testing, mutation, data flow, equivalence, reliability		

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
Program mutation is a new approach to program testing, a method designed to test whether a program is either correct or radically incorrect. It requires the creation of a nearly correct program called a mutant from a program p. An adequate set of test data distinguishes all mutants from P by comparing the outputs. Obviously, an equivalent mutant, which performs identically to P, produces the same outputs as those of P. Thus, for adequate data selection, it is desirable that an equivalent mutant be excluded from the testing process. For this purpose, the system equivalence command has been implemented as an equivalent mutant detector.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

unclassified

unclassified

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

As yet, the command has not been automated. Automatic detection by this command is implemented here as an application of data flow analysis. Algorithms and implementation techniques of data flow analysis are described. Also its application as an automatic detector is described.

ATE

MED

8